Exploiting Implicit Parallelism in Dynamic Array Programming Languages

Shams Imam Vivek Sarkar
Rice University
{shams/vsarkar}@rice.edu

David Leibs Peter B. Kessler
Oracle Labs
{david.leibs/peter.b.kessler}@oracle.com

Abstract

We have built an interpreter for the array programming language J. The interpreter exploits implicit data parallelism in the language to achieve good parallel speedups on a variety of benchmark applications.

Many array programming languages operate on entire arrays without the need to write loops. Writing without loops simplifies the programs. Array programs without loops allow an interpreter to parallelize the execution of the code without complex analysis or input from the programmer.

The J programming language includes the usual idioms of operations on arrays of the same size and shape, where the operations can often be performed in parallel for each individual item of the operands. Another opportunity comes from Js reduction operations, where suitable operations can be performed in parallel for all the items of an operand. J has a notion of verb rank, which allows programmers to simplify programs by declaring how operations are applied to operands. The verb rank mechanism allows us to extract further parallelism.

Our implementation of an implicitly parallelizing interpreter for J is written entirely in Java. We have written the interpreter in a framework that produces native code for the interpreter, giving good scalar performance. The interpreter itself is responsible for exploiting the parallelism available in the applications. Our results show we attain good parallel speed-up on a variety of benchmarks, including near perfect linear speed-up on inherently parallel benchmarks.

We believe that the lessons learned from our approach to exploiting data parallelism in an interpreter can be applied to other interpreted languages as well.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

General Terms Languages, Performance

Keywords Array Programming, Implicit Parallelism, Data Parallelism, AST Interpreter, APL, J, Truffle

ARRAY'14, June 13, 2014, Edinburgh, UK..
Copyright © 2014 ACM 978-1-4503-2937-8/14/06…$15.00.
http://dx.doi.org/10.1145/2627373.2627374

1. Introduction

Exploiting parallelism is important to improve the performance of applications, as more hardware becomes available to more programmers. A number of software constructs are used to allow programmers to specify *explicit* parallelism, either on shared-memory multi-processors, or across clusters of separate machines [18, 19, 21, 22].

Experience has shown that writing explicitly parallel programs is inherently more difficult than writing sequential programs. Array programs offer an alternative: *implicit* parallelism. With implicit parallelism, the language implementer is responsible for discovering and exploiting parallelism. Implicit parallelism relieves the programmer of the need to code the details of the parallelization. It also denies them the opportunity to write incorrect parallelizations, or to hamper the language implementation, or to neglect to parallelize some parts of their code. As a result, implicit parallelism results in a substantial improvement of programmer productivity by enabling writing of correct code quickly. Another benefit of implicit parallelization is that legacy code can be run on parallel hardware without rewriting the programs.

The thesis of this paper is that array programming languages with adequate richness expose opportunities for implicit data parallelism and simplify the problem of writing array-oriented parallel programs. As we describe in Section 3, we achieve implicit parallelism of array programs by exploiting the data parallel property around the notions of function rank [1], vector operations based on scalar operations, and reduction operations involving associative operators.

We use J, a member of the APL family, as an example of a high-level array programming language [8, 10]. Our implementation is an Abstract Syntax Tree (AST) interpreter for J written using the Truffle framework [23] in pure Java™[1], and our parallel runtime uses Java's ExecutorService. Our implementation is sound in that it does not alter the semantics of existing J constructs.

The main contributions of this paper are:

- Identification of data parallel opportunities in array programming languages such as J based on rank agreement on function application, vector operations based on scalar operations, reduction operations involving associative operators, *etc.* In contrast, previous work in other array languages has focused on parallelizing variants of `for` loops in array programs such as Matlab [4] and Single Assignment C [6].
- A new implicitly parallelizing interpreter for the J programming language written entirely in Java. This in stark contrast to interpreters for dynamic languages, like Python and Ruby, that are not multithreading-friendly due to the use of global locks.

[1] Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

- Empirical evaluation of the performance of our interpreter on various programs. Our results show we can attain near perfect linear speed-up with increasing number of processors on some benchmarks, while other programs which are inherently sequential (i.e. do not expose implicitly parallel constructs for us to exploit) run without a major performance penalty.

2. Background

2.1 J

J is an array-oriented language developed in the early 1990s by Kenneth Iverson and Roger Hui [8, 10]; it has an active community of users and is used for teaching and in projects at various companies. J's strength is in the mathematical, statistical, and logical analysis of arrays of data. J uses terms from English grammar to describe the elements of the language: data are called *nouns*, operators are called *verbs*. Higher level operations include *adverbs* and *conjunctions*. Adverbs take a verb as an operand and produce a new verb: for example, one can take the verb for addition and apply an adjective to produce a verb for summation. Conjunctions take two verbs as operands and produce a new verb by various kinds of composition. J has automatic allocation and deallocation for storage. J is a functional language: operations take operands and produce results, but do not modify their operands.

All nouns in J have *rank*, which is the number of dimensions needed to define their *shape*. Scalar values have rank 0. Vectors have rank 1. Matrixes have rank 2. And so on. All verbs in J have ranks, which describe the ranks of the operands on which they operate. Verbs which operate on scalars have rank 0, those that operate on vectors have rank 1, and so on. A dyadic verb has a rank with two elements: one for its left operand and one for its right operand. All the built-in verbs have defined ranks. All user-defined verbs have infinite rank so that they can treat their operands as they will. There is a conjunction to define a verb with a specified rank from an existing verb.

J codifies the idea of *rank agreement* which sets out a simple, consistent rule for how to a verb uses its operands. Application of a rank-k verb to an operand of rank greater than k is defined as the independent application of the verb to each rank-k fragment of the operand, with the final result being formed by merging the individual results. The rank of the final result need not be the same as the rank of an input operand.

Rank agreement can be defined to be the following three step model [15]:

- *Operand slicing, and pairing*: Breaking up the actual operands into nouns with smaller rank so that they agree with the advertised verb rank. For dyadic verbs, corresponding fragments from the input nouns are then paired up.
- *Evaluation on fragment pairs*: The verb is then repeatedly applied on each of the fragments to produce intermediate results.
- *Coalescing of intermediates*: The intermediate results are then merged to form a single noun with a larger rank. Merging needs to account for the case where the fragments are of different ranks, shapes, and types.

For example, multiplication is a rank 0 verb, so when evaluating `2 * 3 5 7` the right hand operand has rank 1 and so is sliced into rank 0 elements (the scalar values `3`, `5`, and `7`), and each slice is paired with the scalar `2` that is the left-hand operand. The evaluations of the application of the rank 0 verb to the rank 0 operands produces intermediate results (which happen to be of rank 0): `2*3`, `2*5`, and `2*7`. The intermediate results are assembled into the final result, a rank 1 vector `6 10 14`.

Slicing and coalescing of nouns and intermediates, respectively, are considered to be part of the fundamental J mechanics and are outside the scope of any specific verb. The rank agreement logic and use of slicing and coalescing is a means of implicit looping in J. We will see in Section 3 that the concept of function rank is fundamental to array parallelism in J.

2.2 Truffle

Truffle is a novel framework for implementing managed dynamic languages in Java [23] by writing AST interpreters. With the Truffle framework, the AST can be modified during runtime to incorporate type feedback to specialize the AST nodes. The ability to replace one node with another, at runtime, enables type specialization of nodes and improves the execution speed of the AST interpretation. Node replacement is based upon assumptions, if the assumption becomes invalid, a node replaces itself with a more general implementation using the newly-found information. As a result the AST keeps modifying itself with dynamic runtime information and this helps in increasing the performance of the interpreter. The Truffle JIT compiler relies on the assumptions and node types to resolve the virtual dispatch overheads from polymorphic call sites opening up opportunities for improved runtime compilation of the interpreter, leading to better performance of the language implementation. Truffle uses the static typing and primitive data types of Java elegantly to avoid the cost of boxed representations of primitive values in dynamic programming languages.

3. Parallelization Opportunities

We believe that an array language like J can help in the writing of efficient implicitly parallel programs. The semantics of J gives the meanings of expressions, but does not dictate how those expressions are to be evaluated. Array programming principles and higher-order constructs in J programs expose copious opportunities for parallelism. In this section, we discuss our techniques to take advantage of three implicit parallelism opportunities: function rank; vector operations on scalar values; and reduction operations that use associative operators.

3.1 Rank Agreement

The concept of function rank is fundamental to array parallelism in J. As mentioned in Section 2.1, when the rank of an operand does not agree with the rank of a verb, the operand needs to be sliced into fragments before applying the verb to each individual slice. For example, rank agreement allows a programmer to apply a verb to slices of a multidimensional array. In J semantics, the order in which the verb is applied to each slice is undefined and there are no side effects a programmer should depend on. This means that computations of intermediate results are inherently non-interfering and can be exploited by traditional data parallelism techniques. With verbs formed by composition, rank agreement logic may be triggered while evaluating the verb on the different slices. This gives rise to opportunities for nested parallelism as in divide-and-conquer style algorithms.

To fully exploit these properties, we have to maximize useful parallel computation and minimize the serial bottlenecks in slicing the operands and in coalescing intermediate results. We minimize slicing overheads by avoiding any copying while creating slices. Coalescing the results often involves copying data from intermediate results into the final result. In scenarios where the shape of the resulting noun can be computed in advance (*i.e.*, before any of the intermediate results become available), we attenuate the serial bottlenecks by *a*) performing the copy in parallel, or *b*) completely eliminating copying overhead by directly targeting the intermediates to their slice location in the result . As a result we can combine steps two and three from the three step model of rank agreement.

3.2 Scalar Verbs

Scalar verbs are verbs with a rank of zero. These verbs operate on individual elements of their operand nouns. The calculation of every individual result element is independent of the others. In other words, application of a scalar verb to array operands represents a vector operation and all scalar verbs present instances of fine-grain parallelism. These can be easily supported on multicore processors by chunking the workload evenly among the available processors. The result array can be preallocated and the result filled in place. Most scalar verbs in J produce a scalar value when applied to scalar operands. In such scenarios the rank agreement logic is simplified and the three step model can be replaced by a simpler scheme where the shape of the result is known in advance. This allows us to directly populate the result array and avoid all copying overheads during the coalescing stage. The specialization of the different stages also reduces effects of the serial stages that can limit parallel performance due to Amdahl's law [7].

3.3 Operand Promotion

When an operand to a verb must be paired with more than one slice of the other operand, the rank of the operand is increased by replicating it to have the number of elements required for the pairings. By recognizing this case we can apply the parallelization techniques described above without replicating the shorter operand.

3.4 Reduction Operations

Reduction operations are formed using the `insert` adverb in J. The insert adverb modifies a dyadic verb and uses this verb to operate on cells of a higher-rank noun in right-to-left order to *reduce* the cells to a single value. Reduction operations can be parallelized only when the verb being applied is associative. We do not require the dyadic verb to be commutative to extract parallelism in the computation since we respect J's right-to-left ordering while evaluating the partial results.

4. Truffle-J Interpreter and Optimizations

Interpreters are a popular choice for execution engines in many virtual machine (VM) architectures because they are simple to implement. We use tree-based representations of the program constructs and the interpreter walks these trees to execute instructions. The Truffle-J interpreter is written in pure Java using the Truffle framework [23]. In this section, we describe some of our optimizations that enable us to execute J programs on our interpreter efficiently in terms of execution time performance. These optimizations include our choice of data structures, tree-rewriting specializations, minimization of temporary array creation, and an efficient parallel runtime.

4.1 Array Data Structure

The choice of data structure to represent arrays dramatically affects the performance of the interpreter. Our choice, called StructA, represents a multidimensional array as a flattened, contiguously allocated one dimensional array of values - we call this the data ravel. This representation is similar to the representation used in the JSoftware interpreter [9] and we claim no novelty in this representation. A StructA provides a read-only view into the data ravel it encapsulates. This choice is facilitated by the fact that arrays in J are immutable. Arrays in J are always rectangular and accesses to arrays are performed by slices. Using the flattened representation, elements of array slices are contiguous in memory and can benefit from caching. This representation comes at the cost of explicit computation of the unidimensional index from multidimensional indices. However in J, such accesses to elements almost always[2] occur on entire arrays. Modern compilers, during just-in-time compilation of the interpreter code, can extract loop invariant code snippets and thereby reduce the overhead of index computation.

In addition to a pointer to the data ravel, a StructA also contains metadata for the array. This metadata includes the shape of the array which gives the dimensions for each rank; an offset index into the data ravel; and the stride length. This representation allows the sharing of the same data ravel by multiple array instances, the constant time creation of slices of arrays, and a constant time promotion of smaller rank arrays into larger rank arrays, for example during operand promotion.

4.2 Truffle-based Node Specialization

Truffle relies on incorporating type information to speed up executions of programs in dynamic languages. Truffle allows an AST to incorporate type information during execution and rewrite itself by introducing guards. Thus, the new AST is more specialized for its input arguments and can execute faster by avoiding boxing and unboxing costs for example. When the types for input arguments do not match, the guards fail and further specialization or generalization of the AST is triggered. When the rewrite orderings do not have cycles, the AST reaches stability. This stable AST can then be optimized by JIT-compiling the interpreter to improve the performance of the interpreter. Our node specializations in Truffle-J include the following: *a*) type specializations for scalar ints, scalar doubles, arrays of ints, or arrays of doubles; *b*) macro expansions for many J constructs such as forks, hooks, conjunctions; and *c*) verb specializations for scalar verbs.

4.3 Verb Fusion and Minimizing Temporary Array Creation

In the traditional J interpreter (and most existing APL implementations), a verb is applied only after its operands have been fully evaluated. This technique can lead to creation of spurious intermediate arrays and affects performance due to increased allocation and garbage collection overhead. In Truffle-J, we dynamically discover new scalar verbs after macro expansion of different language constructs. Figure 1 illustrates the use of this technique more clearly where a J expression (the body of `termFunc` is fused into a *larger* scalar verb. By fusing multiple verbs in the AST sub-trees, we minimize the number of temporary arrays created and improve performance of the interpreter. This technique has been used successfully in the past in statically compiling APL programs [2]; our approach differs in that we discover such opportunities dynamically in addition to the opportunities present in tacit forms of J expressions. Creation of the new fused scalar verbs increases the amount of work done in a unit of computation thereby improving parallel performance by reducing the ratio of overhead introduced during scalar verb parallelization.

```
NB. Compute sum of k^(-0.5) for k from 1 to 10000001
sum =: 3 : ‘(+/) y’
termFunc =: 3 : ‘((_0.5&(^~))@>:) y’
sum (termFunc i. 10000000) NB. 6323.1
```

Figure 1. The body of `termFunc` is effectively a scalar verb which performs the following operations on each element of its input noun y: firstly it increments the value to say `y1` (verb `>:`), secondly it flips the operands to the power function (adverb `~`), and finally it actually computes the value $\texttt{y1}^{-0.5}$ (verb `^`). Without the scalar verb fusion, each of the verbs would need to be applied individually with temporary array creations at each step.

[2] The exception being when the `from` verb is used to select individual elements from a noun

4.4 Parallel Runtime

The key to exploiting parallelism is finding independent subproblems to be solved. We discussed such opportunities in Section 3. We use a task parallel runtime to provide support for parallelism in our interpreter. Since we advocate implicit parallelism, it is the responsibility of the runtime to express our computation in terms of tasks, to create the tasks, to schedule their execution, and to synchronize at the completion of their execution.

Nested Fork-Join Parallelism We use a task parallel runtime to provide support for parallelism in our interpreter since all forms of parallel expression can be broken down into task parallelism. Exploiting parallelism in rank agreement of verb application, vectorization in scalar verb application, etc. all naturally fit into the fork-join model. Each spawned task can transitively discover more opportunities for parallelism and spawn more tasks thus giving rise to nested fork-join parallelism. The parallel runtime in Truffle-J supports task parallelism via the `ExecutorService` framework and creating handles to futures to track task completions at join points. There are two main problems with this scheme: excessive parallelism can lead to the creation of large numbers of tasks, and join points introduce thread-blocking operations and limit scalability and performance.

We limit the excessive creation of tasks by tracking the number of available worker threads at each possible parallelization opportunity. If no worker threads are idle, then the fragment is executed sequentially without creating parallel tasks. On the other hand, when worker threads are available, the amount of work is split as evenly as possible across the number of available workers. Thus, we can dynamically load balance the work at any point across the number of available workers.

Thread-blocking operations at join points block a thread and limit the number of available threads that can participate in executing tasks in parallel. In the parallel runtime, the thread submitting the tasks into the work queue also attempts to execute tasks. As a result, any given task is either executed by the thread that created it (without pushing it onto the queue) or is executed by an independent worker thread that pops the task from the work queue - but not both by guarding its execution around an atomic flag. This scheme ensures when a join point is reached, either all tasks have completed execution or are being actively executed by some other worker thread thus minimizing the wait time at the join point. In addition, the above scheme to avoid excessive task creation ensures there can be no starvation due to all worker threads becoming blocked.

Parallelization of Truffle ASTs in the Interpreter The Truffle framework by itself does not support parallelization and defers the responsibility of introducing parallelism and any necessary synchronization while mutating the ASTs to the guest language implementers. Cloning of ASTs is a common operation in Truffle-based languages, we exploit this feature and rely on creating clones of an AST node per thread to execute a given fragment of a program in parallel. This avoids the need for synchronization while tree rewrites are triggered as any given AST is only being manipulated by a single thread and there can be no data races.

4.5 Truffle-J - Language Completeness

Our implementation effort was mainly driven by the constructs used in the benchmarks we chose to run on the interpreter, as such we only support the int and double datatypes. The J specification defines a list of primitive operations; the Truffle-J interpreter completeness with respect to these primitives is as follows: *a*) Verbs (80 out of 132), *b*) Adverbs (8 out of 18), *c*) Conjunctions (11 out of 30). There are no technical challenges restricting our support for missing language constructs, other data types, or missing primitives.

Name	Source, Computational Feature	JSoftware	Truffle-J	Speed-Up
BlasLevel1a-100M	Ourselves, Linear Algebra	23308.10	1096.51	21.257
BlasLevel2a-5K	Ourselves, Linear Algebra	495.57	827.07	0.599
BlasLevel3a-1K	Ourselves, Linear Algebra	1247.41	3322.80	0.375
BlasLevel3b-1K	Ourselves, Linear Algebra	1110.88	3343.50	0.332
BlasLevel3c-1K	Ourselves, Linear Algebra	1132.88	3485.81	0.325
CodeGolfDigit-25	Ourselves, Scalar arithmetic	18100.50	9888.97	1.830
GameOfLife-2K	C. Jenkins, Stencil Computation	14661.50	10456.06	1.402
Josephus-8M	JSoftware, Scalar arithmetic	18052.80	11368.65	1.588
MatrixInverse-11	JSoftware, Linear Algebra	498176.50	10752.70	46.330
MatrixMult-1000	JSoftware, Linear Algebra	1100.00	3308.89	0.332
MatrixPower-500x16	JSoftware, Linear Algebra	2694.53	7030.81	0.383
MaximalClique-1K	JSoftware, Graph Algorithm	270.98	10280.86	0.026
MaxInfixSum-100K	JSoftware, J adverbs	46.21	16402.84	0.002
MergeSort-16K	C. Jenkins, Array indexing	393.27	2986.02	0.132
PartialSums1-100M	JSoftware, Arithmetic Series Sum	7961.71	34060.59	0.234
PartialSums2-100M	JSoftware, Geometric Series Sum	37975.50	8178.64	4.643
PartialSums3-100M	JSoftware, Inverse quadratic series	66471.60	2860.54	23.237
PartialSums4-50M	JSoftware, Flint Hills series	59824.30	23941.89	2.499
PartialSums5-50M	JSoftware, Cookson Hills series	58577.40	24164.61	2.424
PartialSums6-100M	JSoftware, Harmonic series	26421.60	1453.80	18.174
PartialSums7-100M	JSoftware, Riemann Zeta series	38830.90	1582.42	24.539
PartialSums8-100M	JSoftware, Alternating series	65426.40	9799.50	6.677
PartialSums9-100M	JSoftware, Gregory series	82651.80	10567.65	7.821
PiComputation-20M	C. Jenkins, Pi Series Sum	10688.50	1675.49	6.379
PrimePoly-200	JSoftware, Scalar arithmetic	28294.90	32849.58	0.861
ProjectEuler1-100M	JSoftware, Scalar arithmetic	77211.30	5981.71	12.908
Rank0Verb-100K	Ourselves, Scalar Arithmetic	13900.00	370.75	37.492
SumReduceInt-250M	Ourselves, Series Sum (int)	300.00	838.88	0.358
SumReduceDbl-250M	Ourselves, Series Sum (double)	36604.60	856.79	42.723

Table 1. List of benchmarks used with their sources and the type of computations they represent. The table is sorted alphabetically by the name of the benchmark. Sequential results for benchmarks when run on our interpreter (Truffle-J) and the JSoftware interpreter. Both interpreters are running the same programs. Execution times are reported in **milliseconds**. A speed-up of greater than 1 indicates that the Truffle-J version performed better than the JSoftware version. In the benchmark names, the suffix represents the input data size. E.g. 20M represents an input of 20 million, 2B for 2 billion, 100K for 100 Kilobytes, etc.

5. Experimental Results

We compare the performance of the Truffle-J interpreter with the J interpreter available from JSoftware [12] (version J801). The JSoftware interpreter is implemented in C and has been in development since 1989 by smart and dedicated people [9]. It uses pattern matching techniques to find phrases in J, often called idioms, and interprets them with code tuned to handle such special cases efficiently. Perlis refers to these idioms as mini-operations and provides a comprehensive list in [17]. Our implementation, currently, does no pattern matching on idioms and executes all operations using the same consistent rules.

5.1 Benchmarks

Most of the J programs we use to test the performance of our interpreter are from *real* J programmers, the code for these are available from the JSoftware website which contains extensive documentation and tutorials on J. As shown in Table 1, these benchmarks perform a wide variety of computations. While analyzing the results, it is important to remember that these programs were written as purely sequential code on the JSoftware interpreter with no explicit intent to make them parallelization-friendly.

5.2 Results: Sequential Performance

Methodology The sequential results for the benchmarks were obtained by running on a 4-core Intel Core i7 2.4 GHz system with 8 GB memory. Each core had a 32 kB L1 cache, a 256 kB L2 cache, and a 6 MB L3 cache. The software stack includes Mac OS X 10.7.5, JSoftware interpreter 8.01, Java Hotspot JDK 1.7.0_17, and our Truffle-J interpreter. Each benchmark used the same JVM configuration flags (-XX:+UseBoundThreads -Xmx48G -Xms48G -Xmn32G -XX:+PrintGC -XX:+UseParallelGC) and was run for six iterations, the arithmetic mean of the last five execution times (because the first run triggers tree rewriting specializations and primes the code for JIT compilation) are reported.

We present the sequential results in Table 1 to show that our interpreter has competitive serial performance with respect to the JSoftware interpreter available from JSoftware. Since our data structures use Java primitive arrays, we never allocate more space than is required by the array, we minimize intermediate temporary array creations, and we do not have large garbage collection pauses either. Our interpreter performs poorly on a few benchmarks where the pattern matching techniques from the JSoftware interpreter handles idioms efficiently by implementing *special* logic. A thorough list of idioms supporting such special logic in the JSoftware interpreter is available at [13].

5.3 Results: Parallel Performance

Methodology The parallelization results for the benchmarks were obtained by running on a SPARC T5-8 Server [16]. The machine consists of 8 processors at 3.6GHz, 16 cores per processor, 8 threads per core, for a total of 1024 threads; and 4TB of physical memory. Each processor has 4 memory controllers, a 16KB instruction cache, a 16KB L1 data cache, a 128KB L2 cache, and an 8MB L3 cache. The software stack includes Java Hotspot JDK 1.7 and our Truffle-J interpreter. Each benchmark used the same JVM configuration flags (-Xmx384G -Xms384G -Xmn320G -XX:+UseParallelGC -d64) and was run for six iterations, the arithmetic mean of the last five execution times (first run triggers tree rewriting specializations and primes the code for "JIT" compilation) are reported. Table 2 summarizes the results of running the benchmarks on 1, 2, 4, 8, 16, 32, 64, and 128 worker threads. Note that the machine used tests scalability and is different from the one used to compute the sequential results as we were unable to install the JSoftware interpreter on this machine. The numbers numbers from this table should not be compared to the Sequential Performance table (Table 1).

The benchmarks vary in the amount of parallelism that is actually available. Quite a few of the benchmarks show over 14$\times$ speedup on 16 cores, over 20$\times$ speedup on 32 threads, over 40$\times$ speedup on 64 threads, and over 50$\times$ speedup on 128 threads. We believe these are impressive results given the wide variety of benchmarks and the variance in the amount of parallelism available from the benchmarks. The results also show that our implementation scales as we increase the number of worker threads while working in the managed JVM runtime. The set of benchmarks can be split into three tiers: speedup less than 20$\times$, speedup between 20$\times$ and 50$\times$, and speedup greater than 50$\times$ while running on 128 worker threads.

The MaximalClique benchmark is effectively a sequential application (does not expose any parallelization opportunities that our interpreter exploits) since the speedup hovers around 1, adding worker threads doesn't change the execution times much. In the CodeGolfDigit benchmark, our interpreter is extracting all available parallelism from the benchmark which shows a speedup of 6$\times$, increasing the number of worker threads from 16 to 128 doesn't show further speedup. Some benchmarks involve rearranging or merging arrays (e.g. CodeGolfDigit, GameOfLife, and Mergesort) are memory-bound and show limited improvement after parallelization. Some other benchmarks (e.g. BlasLevel, CodeGolfDigit) flatline at specific values meaning we are exploiting all available parallelism using our techniques. In some of the other benchmarks, creation of temporaries introduces extra GC overhead and limits parallel performance.

Benchmarks in the middle tier of the table (e.g. MatrixPower, MaxInfixSum, SumReduceInt, etc.) have small running times on 128 worker threads. Most of these benchmarks start out with speedup similar to benchmarks from the third tier before the speedup tails off. We believe these benchmarks are affected by having the low computation to parallelization overhead ratio and this negatively affects their scalability.

Benchmarks in the third tier, such as MatrixMult, SumReduceDbl, BlasLevel3 (Matrix), show excellent parallel speed-up. The series sum benchmarks (PartialSums*) also show good parallel speed-up. The best of the benchmarks reach a limit of over 60$\times$ speedup relative to sequential on 128 workers, this is equivalent to having 0.6% serial computation using Amdahl's Law. It seems, even this tiny fraction is limiting our parallelism. One serial part of our computation happens at the barriers (where we wait for each parallel iteration to complete) at the end of each loop either during rank agreement, parallel execution of scalar verbs, or the second phase in a reduction. We feel, this is a very small fraction of the computation that is serial given all the language rules we have to follow.

6. Related Work

The work most closely related to our attempt at parallelizing J is by Christopher Jenkins [11]. Jenkins introduces explicitly parallel constructs for the rank conjunction and the reduction adverb around user-specified associative verbs to statically define parallelism opportunities in a given program. In contrast, we implicitly parallelize existing J programs without introducing new constructs. Our approach subsumes parallelism that comes from rank agreement while also automatically deciding whether a reduction operation can run in parallel by determining whether the verb is associative. Jenkins' Scala interpreter had "underwhelming results", showed very little speedup and often showed slowdowns, especially with large data sizes. Hence, we do not include these comparison numbers in our experimental results (Section 5).

Automatic parallelization to exploit control and data parallelism of APL programs during compile time has been previously attempted by Wai-Mee Ching [5]. Like our approach they do not achieve parallel speed-up when programs are inherently sequential. Unlike in our approach however, the compile time strategies cannot dynamically load balance the available work and minimize creation of spurious parallel tasks. We also address various optimization opportunities to exploit scalar verb inlining and minimizing temporary array creation.

Bernecky also addresses static compilation and automatic parallelization of APL programs in APEX [3]. APEX is an APL-to-SISAL compiler that generates high-performance, portable, parallel code. APEX uses data flow analysis to statically deduce facts regarding the type and rank of each array created in the APL program to exploit the SISAL compiler's capabilities for loop fusion and copy elimination. We discover our opportunities for fusion, copy elimination, and temporary avoidance at runtime. In addition, the APEX compiler represents arrays as vectors of vectors and its storage entails substantial run-time overhead. Our representation of arrays uses contiguous memory blocks and avoids array copying during coordinate mapping operations such as transpose and reversal.

Chunked arrays have been used in ZPL [20], designed from scratch as an implicitly parallel programming language. ZPL's fundamental concept is that of the region - an index set of a specified shape and size. Regions are used to declare parallel arrays and provide index sets for looping mechanisms. In contrast, our approach of parallelizing J relies on regular multi-dimensional arrays which are *chunked* dynamically at runtime efficiently to exploit parallelism.

Table 2. Parallel results for benchmarks when run on our interpreter (Truffle-J) categorized by the scaling of their speedup as numbers of cores increases. The three categories can be interpreted as benchmarks that show poor (or no) speedup, little speedup, and good speedup. The Serial column represents the execution time while running the interpreter in sequential mode, while the Parallel-K columns represent the execution times while running the Truffle-J interpreter with parallel mode enabled and with K worker threads. Execution times are reported in **seconds** with one decimal place accuracy, the speed-ups compared to the sequential execution time as the baseline are reported with two decimal places accuracy.

Benchmark	Serial	Parallel-1		Parallel-2		Parallel-4		Parallel-8		Parallel-16		Parallel-32		Parallel-64		Parallel-128	
MaximalClique-1K	19.3	19.3	1.00	19.8	0.97	20.9	0.92	21.5	0.90	19.5	0.99	18.9	1.02	18.9	1.02	19.4	0.99
CodeGolfDigit-25	26.3	26.3	1.00	16.5	1.60	9.1	2.91	5.4	4.91	4.3	6.13	4.7	5.63	4.2	6.24	4.5	5.82
GameOfLife-3K	75.4	76.3	0.99	47.3	1.59	24.4	3.09	14.6	5.16	11.5	6.55	10.5	7.21	12.9	5.85	10.8	7.00
MergeSort-32K	25.8	25.8	1.00	13.4	1.93	8.1	3.17	5.1	5.10	4.2	6.18	4.0	6.52	3.1	8.24	3.1	8.43
BlasLevel2a-25K	40.3	40.3	1.00	22.0	1.83	12.2	3.30	7.3	5.49	4.4	9.05	3.4	11.88	3.0	13.56	2.9	13.67
BlasLevel1a-2B	36.2	36.2	1.00	26.9	1.35	13.4	2.70	6.0	6.06	3.9	9.41	2.7	13.61	2.1	16.86	2.3	15.88
Josephus-8M	26.1	26.1	1.00	18.3	1.43	8.7	3.01	4.8	5.38	2.7	9.70	2.0	13.04	1.6	16.49	1.3	19.43
PiComputation-200M	16.6	16.6	1.00	8.5	1.95	4.4	3.73	2.4	6.91	1.4	12.06	1.2	14.38	0.7	22.88	0.8	19.68
PartialSums7-400M	13.8	13.8	1.00	7.1	1.94	3.7	3.72	2.0	6.85	1.2	11.62	0.8	17.95	0.6	23.76	0.5	25.33
PartialSums6-400M	12.9	12.9	1.00	6.6	1.95	3.5	3.69	1.9	6.80	1.1	11.55	0.8	16.82	0.6	20.80	0.5	25.40
PartialSums3-400M	14.6	14.6	1.00	7.5	1.94	3.9	3.72	2.1	6.85	1.2	11.77	0.8	17.75	0.6	22.55	0.5	26.73
MatrixInverse-12	259.8	260.4	1.00	140.1	1.85	71.1	3.65	47.4	5.48	24.6	10.58	27.0	9.61	24.9	10.45	7.8	33.52
ProjectEuler1-100M	15.7	15.7	1.00	8.0	1.96	4.7	3.35	2.2	7.12	1.3	11.95	0.8	20.51	0.5	31.69	0.4	36.75
MatrixPower-500x16	14.6	14.6	1.00	7.3	1.99	3.7	3.96	1.9	7.55	1.0	14.96	0.6	23.00	0.4	40.64	0.4	37.26
SumReduceInt-2B	15.0	15.0	1.00	7.6	1.98	3.8	3.98	1.9	7.91	1.0	15.54	0.8	18.67	0.5	29.80	0.4	42.16
Rank0Verb-10M	37.3	37.3	1.00	11.7	3.19	6.0	6.20	5.7	6.53	3.1	12.09	1.8	21.12	1.0	39.05	0.8	46.72
MaxInfixSum-100K	37.2	37.2	1.00	37.2	1.00	20.7	1.80	9.9	3.77	4.8	7.75	2.4	15.71	1.2	31.44	0.8	47.03
PrimePoly-200	64.5	64.5	1.00	59.2	1.09	23.0	2.81	10.9	5.92	6.1	10.61	3.5	18.57	2.0	31.78	1.3	51.13
BlasLevel3c-3K	211.8	219.0	0.97	106.5	1.99	53.4	3.97	28.1	7.55	14.2	14.90	10.7	19.88	4.3	49.36	4.1	51.99
MatrixMult-2500	119.4	119.4	1.00	59.7	2.00	29.9	4.00	15.0	7.98	7.6	15.66	3.8	31.42	2.9	41.49	2.2	54.23
BlasLevel3b-3K	210.8	210.7	1.00	105.6	2.00	53.0	3.98	26.8	7.88	13.7	15.38	10.3	20.45	5.7	37.07	3.8	55.53
SumReduceDbl-2B	15.1	15.1	1.00	7.7	1.97	3.8	3.94	1.9	7.84	1.0	15.46	0.5	30.09	0.3	56.36	0.3	57.14
PartialSums9-300M	42.2	42.2	1.00	21.9	1.93	11.2	3.78	5.7	7.35	3.0	14.02	1.6	25.82	1.0	41.82	0.7	58.56
PartialSums8-300M	38.8	38.8	1.00	20.0	1.93	10.2	3.80	5.3	7.34	2.8	13.97	1.5	25.44	1.0	38.56	0.7	58.80
BlasLevel3a-3K	206.7	206.8	1.00	103.4	2.00	51.6	4.00	25.9	7.97	13.1	15.78	10.2	20.34	5.1	40.42	3.4	61.00
PartialSums1-250M	35.3	35.3	1.00	18.2	1.94	9.5	3.73	4.9	7.26	2.6	13.54	1.4	24.68	0.8	41.97	0.6	61.75
PartialSums4-75M	50.2	50.2	1.00	25.8	1.94	13.0	3.87	6.5	7.68	3.3	15.11	1.8	28.48	1.0	48.69	0.8	66.02
PartialSums5-75M	50.7	50.7	1.00	26.2	1.93	13.2	3.84	6.7	7.61	3.4	14.96	1.8	27.78	1.1	47.62	0.8	66.09
PartialSums2-250M	55.8	55.8	1.00	29.3	1.90	14.8	3.77	7.5	7.43	3.9	14.41	2.1	27.17	1.3	44.41	0.8	68.39

Single Assignment C (SAC) [6] is a statically-typed strict, purely functional language whose syntax, in large parts, is identical to that of C. An array in SAC is represented by a shape vector which specifies the number of elements per axis, and by a data vector of the array's ravel. This representation of arrays is similar to how we represent arrays via the StructA data structure. SAC uses a statically compiled approach and there is only only one level of parallelism (via the `with`-loop construct), either a *master* thread executes or workers are executing fragments of the same task. In our approach all worker threads can be executing independent pieces of work and acting as a master threads among a subgroup of worker threads (nested fork-join style parallelism).

Repa [14] introduces regular, multi-dimensional arrays in Haskell. It is purely functional and supports reuse through shape polymorphism. Like our approach, it avoids unnecessary intermediate structures rather than relying on subsequent loop fusion, and also supports transparent parallelization. Repa reports similar scaling as our work for the matrix multiplication benchmark up to 64 threads.

7. Summary

Array programming languages with adequate richness expose opportunities for implicit parallelism. Effectively exploiting these opportunities simplifies the task of writing parallel programs. In this paper, we have presented our efforts to implement an implicitly parallel interpreter for J. Our implementation, Truffle-J, is an abstract syntax tree interpreter based on the Truffle framework and is written in pure Java. Our results show we attain good parallel speed-up on a variety of benchmarks, including near perfect linear speed-up on inherently parallel benchmarks until the constraints of Amdahl's law start to dominate. We believe that the lessons learned from our approach of exploiting parallelism in an interpreter can be applied to other interpreted array languages as well.

References

[1] R. Bernecky. An Introduction to Function Rank. In *Proceedings of the International Conference on APL*, APL '88, pages 39–43, New York, NY, USA, 1988. ACM.

[2] R. Bernecky. The Role of APL and J in High-performance Computation. In *Proceedings of the international conference on APL*, APL '93, pages 17–32, New York, NY, USA, 1993. ACM. ISBN 0-89791-612-3. . URL http://doi.acm.org/10.1145/166197.166201.

[3] R. Bernecky. APEX – The APL Parallel Executor. Master's thesis, University of Toronto, 1997.

[4] A. Chauhan and K. Kennedy. Optimizing Strategies for Telescoping Languages: Procedure Strength Reduction and Procedure Vectorization. In *In ACM Intl. Conf. on Supercomputing (ICS04)*, pages 92–101, 2001.

[5] W.-M. Ching. Automatic Parallelization of APL-style Programs. In *Conference Proceedings on APL 90: For the Future*, APL '90, pages 76–80, New York, NY, USA, 1990. ACM. ISBN 0-89791-371-X.

[6] C. Grelck and S.-B. Scholz. SAC: off-the-shelf support for data-parallelism on multicores. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, New York, NY, USA, 2007. ACM.

[7] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, July 2008. ISSN 0018-9162. . URL http://dx.doi.org/10.1109/MC.2008.209.

[8] R. K. W. Hui, K. E. Iverson, E. E. McDonnell, and A. T. Whitney. APL\? In *Conference Proceedings on APL 90: For the Future*, APL '90, pages 192–200, New York, NY, USA, 1990. ACM. ISBN 0-89791-371-X.

[9] Hui, Roger. *An Implementation of J*. Iverson Software Inc., 1992. URL http://www.jsoftware.com/jwiki/Doc/An%20Implementation%20of%20J.

[10] K. E. Iverson. The Dictionary of J. *Journal of the British APL Association*, 7(2):99–117, October 1990.

[11] C. Jenkins. Toward a Parallel Implementation of J: Data Parallelism in Functional, Array-Oriented Languages with Function Rank. Computer Science Honors Theses, Paper 30, Trinity University, April 2013. http://digitalcommons.trinity.edu/compsci_honors/30.

[12] Jsoftware Inc. Jsoftware High-performance development platform, . URL http://www.jsoftware.com/.

[13] Jsoftware Inc. Appendix B. Special Code, . URL http://www.jsoftware.com/help/dictionary/special.htm.

[14] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3.

[15] M. Neitzel. Untying the Gordian Knot: Agreement in J. In *Proceedings of the International Conference on Applied Programming Languages*, APL '95, pages 145–153, New York, NY, USA, 1995. ACM.

[16] Oracle. SPARC T5-8 Server. URL http://www.oracle.com/us/products/servers-storage/servers/sparc/oracle-sparc/t5-8/overview/index.html.

[17] A. J. Perlis and S. Rugaber. Programming with Idioms in APL. *SIGAPL APL Quote Quad*, 9(4):232–235, May 1979. ISSN 0163-6006.

[18] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. ISBN 9780596514808.

[19] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM. ISBN 0-89791-710-3. . URL http://doi.acm.org/10.1145/224964.224987.

[20] L. Snyder. The Design and Development of ZPL. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7.

[21] The MathWorks Inc. Parallel Computing Toolbox - MATLAB. URL http://www.mathworks.com/products/parallel-computing/.

[22] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009. ISBN 0596521979, 9780596521974.

[23] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST Interpreters. In *Proceedings of the 8th symposium on Dynamic languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1564-7.